Stateful Container Migration in Geo-Distributed Environments

Paulo Souza Junior, Daniele Miorandi, Guillaume Pierre

Stateful Container Migration in Geo-Distributed Environments

Paulo Souza Junior
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
paulo.souza@inria.fr

Daniele Miorandi
U-Hopper
Trento, Italy
daniele.miorandi@u-hopper.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
guillaume.pierre@irisa.fr

***Abstract*—Container migration is an essential functionality in large-scale geo-distributed platforms such as fog computing infrastructures. Contrary to migration within a single data center, long-distance migration requires that the container's disk state should be migrated together with the container itself. However, this state may be arbitrarily large, so its transfer may create long periods of unavailability for the container. We propose to exploit the layered structure provided by the OverlayFS file system to transparently snapshot the volumes' contents and transfer them prior to the actual container migration. We implemented this mechanism within Kubernetes. Our evaluations based on a real fog computing test-bed show that our techniques reduce the container's downtime during migration by a factor 4 compared to a baseline with no volume checkpoint.**

I. Introduction

Migration is an essential functionality in large-scale virtualized computing platforms. Migrating virtual machines is commonly used by data center managers as an enabler for scheduled server decommission, resource consolidation, disaster recovery, vertical scaling, etc [1]. As many applications are moving from VM-based to container-based infrastructures for reasons of simplicity, performance and cost, similar techniques are becoming necessary in container environments as a fundamental system management tool.

Although early container-based platforms relied on low-level container technologies such as LXC, the increased popularity of container frameworks has resulted from the introduction of high-level tools such as Docker (which provides developer-friendly software development workflows) and Kubernetes (which orchestrates Docker containers at the scale of a cluster or a data center). However, the current container migration techniques are designed at the lowest level only, making them unsuitable for usage in higher-level container orchestration environments.

Container migration is made even more important by the trending usage of container orchestration frameworks such as Kubernetes as a basis for designing "fog computing" environments capable of extending traditional cloud data centers with additional resources located close to the end users [2]–[6]. In this context, container migration may exceed the domain of system management tools and additionally become a primary platform feature allowing operators to migrate a fog computing application from one location to the next to follow the mobility of human end users.

Container migration in geo-distributed fog computing environments creates new challenges [7]. In particular, containers may use data volumes to store their application-level files and databases. In single-datacenter environments, these volumes are typically stored in a separate network-attached storage (NAS) [8]. Upon any VM or container migration within the same data center, storage volumes do not need to be migrated as the new VM or container may simply re-attach the same volume. However, using the same technique upon a geo-distributed container migration would imply that the new container should issue long-distance remote data access, which is likely to negate the performance benefits of any such migration. When migrating a container from one fog computing location to another, it is therefore important to seamlessly migrate its data volumes as well. On the other hand, this operation must be carefully organized in order to minimize the downtime witnessed by external users who may be accessing the container during its migration.

In this paper, we propose to exploit the OverlayFS layered structure of Docker container volumes to migrate the files that are not being actively modified prior to the actual container migration. This significantly reduces the migration downtime as only a small number of "hot" files must be transferred during the downtime. We integrate this migration technique within Kubernetes and ensure that container migration remains transparent to external users. Our evaluation results based on a real fog computing testbed show that this technique reduces the user-perceived downtime during migration by a factor 4 compared to a baseline migration process.

The remainder of this paper is organized as follows. Section II presents the technical background, then Section III discusses related works. Section IV presents the design and implementation of our stateful container migration approach. Section V evaluates it, and finally Section VI concludes.

This work is part of the FogGuru project which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

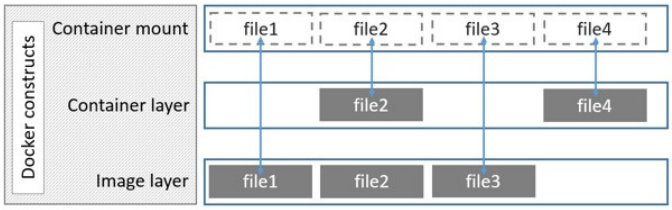

Fig. 1. Layered file system structure.

II. Background

A. Docker

Docker is by far the most popular container management framework in cluster, cloud and fog environments [9]. Docker containers are packaged in the form of *images* which consist of multiple *layers*, where each layer contains a part of container's file system that contains application's binaries, libraries, data, etc. This structure makes it very easy for developers to incrementally define new container images by adding a specialization layer on top of an existing image. Docker supports multiple layered file systems, the default one being OverlayFS.

The same layering strategy is also used to store file system updates performed by the applications after a container has started: upon every container deployment, Docker creates an additional writable top-level layer which stores all updates following a Copy-on-Write (CoW) policy. The container's image layers themselves remain read-only. As shown in Figure 1, although each file system layer is stored separately on disk, the layered file system exposes a single merged "container mount" view of all the layers to the container. The top "container layer" contains every file which was created or updated by the container since its creation. Under it, one or more read-only "image layers" typically store the container's programs, libraries and configuration files. Image layers can usually be fetched from public repositories such as the Docker Hub.

B. Kubernetes

Kubernetes is a container orchestration platform which automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter [10]. It relies on Docker as its default container engine in charge of creating, deploying and running containers in any of the Kubernetes work nodes.

In Kubernetes the smallest software deployment unit is a *pod*, defined as a tight set of logically-related containers and data volumes to be deployed on a single machine. Kubernetes assigns each pod with CPU, memory and disk resources in one available worker node in the system. Figure 2 depicts a pod with two containers and one volume. The choice of which node should host which pod is made by the Kubernetes scheduler [11].

Pods may be stopped and replaced at any moment, for example, upon a failure of any of their containers. To provide pods with a stable network address, Kubernetes defines a *service* as a stable IP address, possibly exposed to external users, which acts as a load balancer between a set of related pods. This means that, upon a pod's migration from one Kubernetes worker node to another, the pod may remain reachable by external end users by simply exposing a stable service IP address which gets dynamically rerouted to the new pod's location.

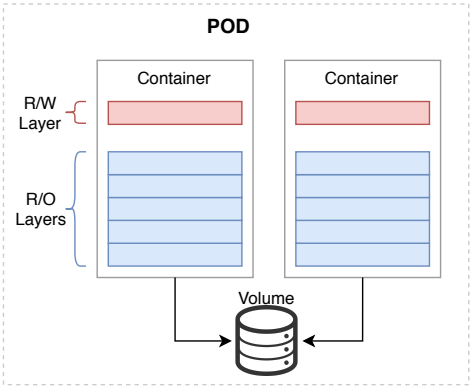

Fig. 2. A Kubernetes pod with two containers and one volume.

In Kubernetes the Docker read-write container layers are explicitly assumed to be ephemeral. When a pod is stopped (a frequent operation in Kubernetes), its container layers are simply discarded. Persistent storage is provided in the form of *volumes*, which are virtual disks available to the containers within a pod [12]. Volumes remain persistent even when a pod is stopped, and they are re-attached when the pod is restarted. Multiple containers inside a single pod may share the same volume and perform any read/write operations simultaneously; this pattern is commonly used in Kubernetes. Upon a geo-distributed pod's migration, it is therefore essential to also migrate the volume's content so the pod's data remains co-located with the containers which access them.

III. Related Work

Numerous studies have focused on various aspects of migration for virtualized environments. We focus here only on techniques dedicated to container migration.

Container migration techniques can be classified along two dimensions. First, container migration may be *stateful* or *stateless*. In stateful migration the system aims to recreate the new container with the same memory, system and disk state as the original one. Stateless migration, on the other hand, does not aim to maintain this state in the new container.

Second, migration may be either *cold* or *live*. Cold migration requires one to stop the container before performing its migration whereas live migration transparently checkpoints the memory state of the container and therefore does not require one to explicitly stop the container. Cold migration usually implies a longer downtime than live migration [7]. We however show in this paper how this downtime may be significantly reduced, even in stateful migration where the container has large disk state.

A number of research work focus on supporting container migration in a fog computing environment. For instance, Follow Me Fog aims at allowing IoT devices to control the timing of container migration in the fog to follow its mobility [13]. However, it focuses mostly on avoiding unnecessary migrations.

When applications are elastic it is possible to exploit replication to perform migration by creating a new application replica followed by stopping the old one [14], [15]. However, this form of migration is necessarily stateless as the new replica does not receive the state of the old one. In contrast, we aim at preserving the container's disk state across the migration.

Performing stateful migration requires one to snapshot the state of the migrated container. Several checkpointing mechanisms may be used, the most popular one being CRIU [16]. CRIU is a kernel module which snapshots the contents of a container's memory pages, open sockets, open files, etc. However, to perform these operations, CRIU needs to stop the running service and reload all the information inside the container to create a new snapshot of the current state. As a result, CRIU-based migration techniques usually implement cold migration [17]–[20]. This creates potentially very large downtime as the entire state must be transfered to the target node before the container can be restarted.

An interesting system is VAS-CRIU, a variant of CRIU which saves the checkpointed state in memory rather than disk [21]. This avoids costly disk operations, but requires large amounts of DRAM instead. It is therefore applicable mostly to powerful datacenter servers with large amounts of memory.

Last but not least, a number of works use a similar approach to ours which exploits the layered file system structure of Docker containers. Ahmed *et al* checkpoint a container's state and its layered file system structure, but use the checkpoint to restart multiple identical containers rather than migrate the checkpointed container [22]. Ma *et al* propose to download the container's read-only image files in the target before migrating the container [23]. We build upon similar ideas, but push them further to also checkpoint the current state of the read-write container layer, which further reduces the container's migration downtime.

IV. System Design

Migrating a pod from one Kubernetes node to another is in principle very simple: one essentially needs to stop the pod, capture and transfer its state to the destination node, restart a new pod based on the transferred state, and adjust the network routing rules to make the new pod available under the same IP address as the old one. However, implementing this in a geo-distributed Kubernetes setup creates a number of challenges. First, if the state is large this simple strategy may imply large downtime while it is being transferred from one fog computing location to another. In this paper we specifically focus on disk state, which may grow arbitrary large over time depending on the pod's activity. Second, one needs to coordinate the different operations to snapshot the pod's state, stop and restart the pod in the correct locations, and re-establish correct routing. We discuss these two challenges in turn.

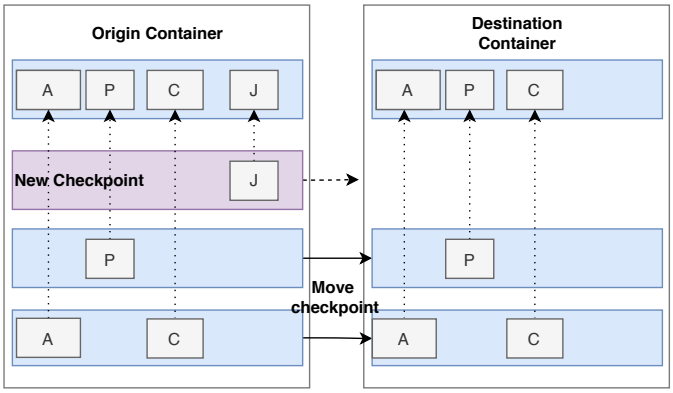

Fig. 3. Checkpointing disk volume checkpoints.

A. Volume checkpointing

Kubernetes pods may be composed of one or more Docker containers, which have access to two distinct forms of disk storage. First, each container has access to its "virtual local disk" in the form of a container image. In this image, the bottom layers are read-only through the lifetime of the container. They can thus be copied to the destination nodes prior to the container migration without incurring any downtime. Since these layers constitute the container's image which can be fetched from a Docker repository, we do not need to copy them explicitly from the source to the destination node but rather rely on Docker's default behavior which fetches these layers from the repository upon any container deployment, unless they are already available in the local image cache.

The top-level read-write container layer captures all the file system updates issued by the container since its creation. Depending on the container's activity it may contain updated versions of the files from the underlying layers (following a copy-on-write policy), and any new file that the application inside the container may have created. However, in Kubernetes this container layer is explicitly assumed to be ephemeral. When a pod is stopped, its container layers are simply discarded. When migrating a pod it is therefore not necessary for us to transfer this part of the state.

Persistent storage in Kubernetes is provided in the form of *volumes*, which are virtual disks available to the containers within a pod. Volumes remain persistent even when a pod is stopped, and they are re-attached when the pod is restarted. Kubernetes supports many types of volumes stored either in the same node as the pod or in a local Network Attached Storage (NAS). In our implementation we use local volumes to maintain proximity between the containers and their disk storage in a geo-distributed environment. However, the same concept also applies to volumes stored in a NAS, which must also be migrated from one NAS to another.

Upon a pod migration we therefore need to snapshot the state of the pod's volume(s) precisely at the time when the pod

has been stopped, and to transfer its content to the destination node before restarting a new pod based on this disk state. A naïve baseline strategy would simply transfer the container layer's content after the old pod has been stopped. However, we show in Section V that this may require long transfer times before the new pod can be restarted.

A better strategy consists of identifying the parts of the container layer which are not currently being modified, and to transfer them prior to stopping the old container. To do this we require that the pod's volume should be formatted using the same layered OverlayFS file system as is used for the container's image. Kubernetes volumes are usually formatted using a non-layered file system such as `ext4`. Note that OverlayFS does not introduce any significant performance overhead compared to `ext4` [24].

When migrating a Kubernetes pod, we use OverlayFS to create a snapshot of the pod's volume(s). Upon this operation, the snapshot content becomes read-only, and a new empty read/write layer is added on top. This operation is transparent for the container's processes which only see the merged file system created by OverlayFS. Transferring the newly created volume snapshot may take time during which new file system updates are issued on the container layer. It is therefore possible to create and transfer more than one snapshot before stopping the container and transferring the last remaining file system updates. Figure 3 illustrates this process.

The different operations required to migrate a container are issued by a migration tool that must be included in the container when it is created in the first place. This migration tool implements the following API:

Init Volume: When a pod starts, Kubernetes mounts its volume(s) as a regular non-layered file system. The Init Volume method checks if the volume already contains the "lowerdir" and "upperdir" directories required by OverlayFS, creates them if necessary, and remounts the volume in the pod's containers using the OverlayFS file system.

Create Checkpoint: This method checkpoints the pod's volume. As a result the current top-level layer of the volume becomes read-only (with the content of the checkpoint), and a new empty read-write layer is created on top to capture any future file system updates. The method then remounts the volume on the fly with the new layer structure.

Copy Checkpoint: This method is called at the origin node upon a pod migration to copy one or more checkpoint layers to the destination node. We assume that the network bandwidth is the main bottleneck in a geo-distributed environment, therefore this method compresses the layers before transferring them.

Receive Checkpoint: This method is issued at the destination node to identify the checkpoint that has been copied, and mount it in the new pod following the same strategy as the Init Volume method.

End Volume: Whenever a pod is stopped because it is being migrated to another node, this method unmounts the volume and migrates the last file system updates that were not yet transferred. It finally releases the volume to be recycled by Kubernetes.

During a pod migration, the first snapshot captures the full state of the volume at the time the migration is initiated. After being snapshot, this content becomes read-only so it can be transferred to the destination node prior to stopping the old container. One or more additional snapshots may be created so that file system updates which took place while transferring the first can be migrated out of the critical path of the container's downtime. The only content which must be transferred during the downtime therefore remains small. Transferring this last file system layer takes place during the downtime after the old pod has been stopped, and before the next one can be started. The different operations must be carefully coordinated to minimize the downtime, as we discuss next.

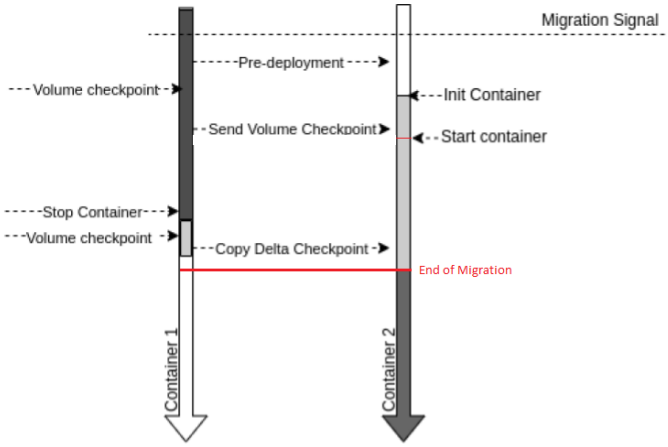

Fig. 4. Pod migration process.

B. Coordination

When a pod migration request is issued, an ephemeral "coordinator" container is created in the origin node, in a new pod, to coordinate the migration. The coordinator has full access rights to the Kubernetes and volume management APIs.

Figure 4 depicts the migration process. Migration begins with an initial request from the user or application manager. When the coordinator starts, it receives the migration parameters, the pod name, the destination node, and the number of requested checkpoints. It then instructs Kubernetes to create a new pod in the destination node by attaching its request for a new pod with Kubernetes labels which constrain its placement on the desired node. The new pod is created with a thin additional image layer which contains an *initializer* script in charge of receiving the checkpoints and setting up the new pod.

The coordinator then creates the first volume checkpoint. This process dumps all the content of the upper layer of the volume to a read-only layer, and creates a new top-level empty layer. The coordinator invokes the Copy Checkpoint function to copy this layer to the destination container. The process may

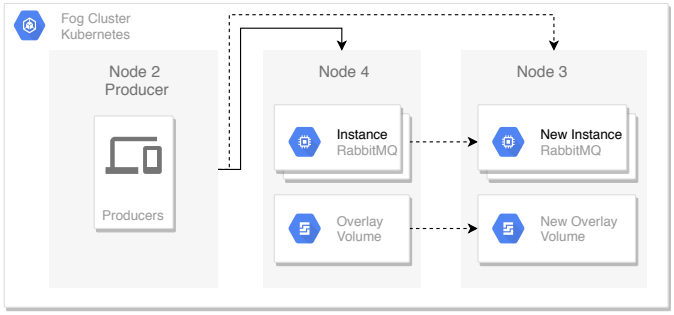

Fig. 5. Experimental setup.

be repeated in case additional checkpoints are required. During this time the origin container keeps running and potentially issuing file system updates in its new top-level layer. The final steps consist of stopping the old container, invoking the End Volume to migrate the last remaining file system updates, and deleting the volume so it can be recycled by the Kubernetes system.

At the destination node, every time a new snapshot arrives, the initializer script is in charge of storing it in a new file system layer and of remounting the volume in the container. After receiving the last file system updates, the initializer invokes the command to start the container's services. It finally updates the pod's labels to indicate to the Kubernetes that it needs to update its load-balancing rules and redirect the network traffic to the new pod instead of the old pod.

As soon as this reconfiguration is completed, the pod migration is finished. The pod's migration remains transparent to its users which keep addressing the new pod using the same IP address as they were using before the migration. They may observe a downtime period during which the pod remains unresponsive, but they need not be aware of any detail about the migration.

V. Evaluation

A. Experimental setup

We evaluate this work using a fog computing testbed composed of five Raspberry Pi (RPI) single-board computers model 3 B+ with quad-core 1.2 GHz CPU, 1GB of RAM and a 32 GB micro-SD storage device. This type of machine is frequently used to prototype fog computing infrastructures [3], [25]–[27]. The RPIs use the HypriotOS Linux distribution, Kubernetes 1.16 and Docker 19.03.5. The experimental setup is illustrated in Figure 5: the first RPI acts as the Kubernetes master node while the other four can run application pods.

We exercise the system using a RabbitMQ 3.7 persistent MQTT service [28] running inside a pod which is migrated during its execution between different machines. This type of queuing service is frequently used in fog computing platforms [29]. RabbitMQ uses 70 MB of storage for its image content, and an extra 400 MB of state in its disk volume to persist messages before they have been fully delivered. RabbitMQ is therefore representative of a demanding IO-intensive fog application.

TABLE I
Evaluation parameters.

# of checkpoints	0, 1, 2
Bandwidth (kbps)	500, 1000, 2000
Pre-pull strategy	Yes, No

We generate a message load using RabbitMQ-PerfTest [30] with one message producer which produces random data in JSON format based on a fixed seed, with 100 messages/s of 128 kB each distributed between two different queues. Messages are sent to RabbitMQ with no consumer to receive them.

In our experiments, we first start the RabbitMQ pod with one container and one 20 GB data volume on Node 4, generate workload with the producer's pod running in Node 2, and then migrate the RabbitMQ pod to Node 3. Each experiment lasts 150 s and the pod migration is issued 60 s after starting the workload.

We compare the performance of our pod migration mechanism with that of a baseline strategy which does not checkpoint the pod's volume. The baseline (thereafter named "0 checkpoint") simply stops the old pod, migrates its entire state to the destination node, and restarts the pod in the new location. Table I shows the tested configurations for the pod migration system. We vary the number of checkpoints from 0 to 1 and 2 checkpoints. We control the available network bandwidth between the worker nodes using the "traffic control" (`tc`) tool available in Linux systems, with 500 kbps, 1000 kbps and 2000 kbps. These values are based on a recent study which highlights the characteristics of today's networking technologies used in fog computing settings [31]. Finally, we test two variants of the migration algorithm which respectively pulls and does not pull the container image layers in the destination node prior to the migration.

B. Message throughput during migration

Figure 6 shows the RabbitMQ message throughput before, during and after migration while processing a workload of 100 messages/s. In this experiment we use a network bandwidth of 2000 kbps and the pre-pull strategy, and vary the number of checkpoints from 0 (baseline algorithm) to 1 and 2. The vertical lines depict the time at which the migration is initiated and completed.

In the baseline algorithm with no volume checkpoint prior to the migration, the message throughput drops to zero immediately after the start of the migration procedure. We observe a downtime of about 40 seconds until the volume has been transferred and the new container is operational again. Immediately after restart the message throughput increases to large values because the message producer delivers all the produced messages which accumulated during the downtime. Finally, the throughput returns to its initial value.

In the next experiment with one checkpoint prior to migration, we observe that the downtime starts several seconds after the migration was initiated. The duration of the downtime is

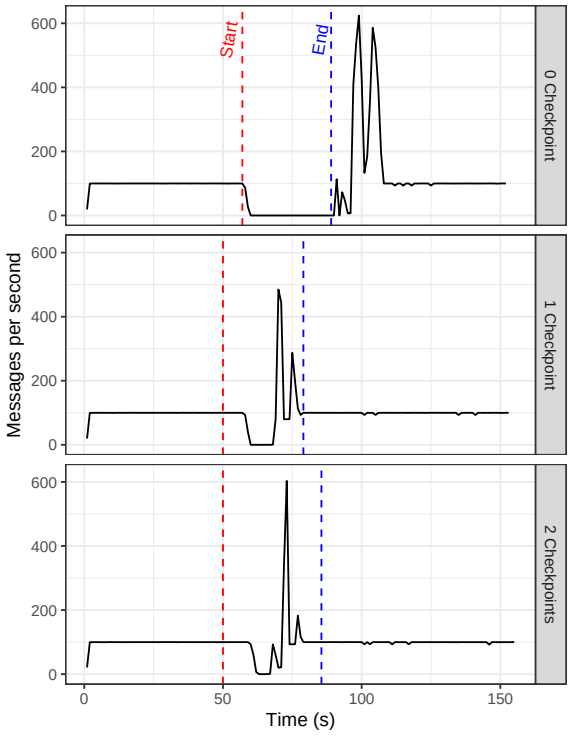

Fig. 6. RabbitMQ's message throughput during migration with 2000 kbps bandwidth, pre-pull strategy, and different numbers of checkpoints.

also noticeably shorter. This is due to the fact that a smaller amount of data needs to be transferred between the concerned nodes during the downtime. Also, we observe that the spike of messages to be delivered after migration is smaller than using the baseline algorithm. This is due to the fact that a smaller number of undelivered messages were buffered during the container downtime. A longer downtime results in a higher number of buffered messages sent when the service becomes available again. In busy fog computing environments this provides an additional benefit to our approach as a shorter downtime creates a smaller backlog of operations which need to be processed after the migration. Finally, when using two checkpoints before migrating the container we observe that the downtime is further slightly reduced.

The performance gains of increasing the number of checkpoints creates a tradeoff between the volume of data to be transferred (which is greater with larger numbers of checkpoints) and the observed container downtime.

We also notice that the presence of the migration tool does not impact the running application's performance, as no interference is visible in the throughput of the application prior to migration. The agent inside the ephemeral container creates the checkpoints using lazy mounting in the filesystem, with minimum impact on the running application container(s).

C. Container downtime

Figure 7 shows the overall container downtime across the full set of experimentation scenarios. We run each experiment 15 times, and report the mean, minimum, maximum, 25th and 75th percentile values.

In all experiments, the performance benefits of checkpointing the pod's volume before migration is evident. We observe a reduction of the downtime by a factor 4 between the baseline with no checkpoint, and our migration technique with one checkpoint. A second checkpoint further reduces the downtime a little, yet at the cost of increased variability between runs. This is due to the fact that RabbitMQ does not write constantly to disk, so the volume of the second checkpoint varies from one run to the next.

We observe that the available network bandwidth does not significantly influence the downtimes. This is due to the fact that the main bottleneck is created by slow disk I/O (the RPIs use micro-SD cards as their only stable storage).

Finally, we observe that pre-pulling the container layers improves performance a little, especially for the situations with low bandwidth. Notwithstanding, migration with more checkpoint provide an extra time to pre-download the content of a container image, it impacts directly in low bandwidth network, however, it does not provide further improvements in scenario where just one checkpoint is used, especially with larger bandwidth. When two checkpoints are used, more time is provided to pre-download the images since the service stays up for more time before it is stopped to be replaced for a new one. It happens because there is a fixed interval between creating a checkpoint after another, which is not arbitrarily defined and can be changed under different circumstances.

D. Resource usage

In a fog computing infrastructure, resources may be scarce. It is therefore important to use them with caution. On the other hand, a container migration necessarily creates an additional burden to the infrastructure.

Figure 8 shows the CPU and memory usage of the two worker nodes respectively hosting the old and the new container during migration, when using zero, one or two checkpoints. These experiments exploit the pre-pull strategy as we know that it provides better performance than its no-prepull counterpart.

Before migration we observe that Node 4 has greater CPU and memory usage than Node 3. This is expected, as Node 4 runs the RabbitMQ pod whereas Node 3 remains idle. When the migration is initiated, the memory usage of Node 3 increases as it receives the checkpoints and writes them to disk. Once this is finished, the memory usage in Node 3 matches that of Node 4 before migration. Conversely, as soon as the old pod gets deleted in Node 4, its memory usage drops very quickly.

We observe that migration is more CPU-intensive in the destination node than the origin node. This is due to the fact that the destination node must start a new container, create a local layered file system, receive checkpoints, and remount the file system. Also, as soon as migration is finished, the RabbitMQ application needs to process all the delayed messages during migration, which generates additional load.

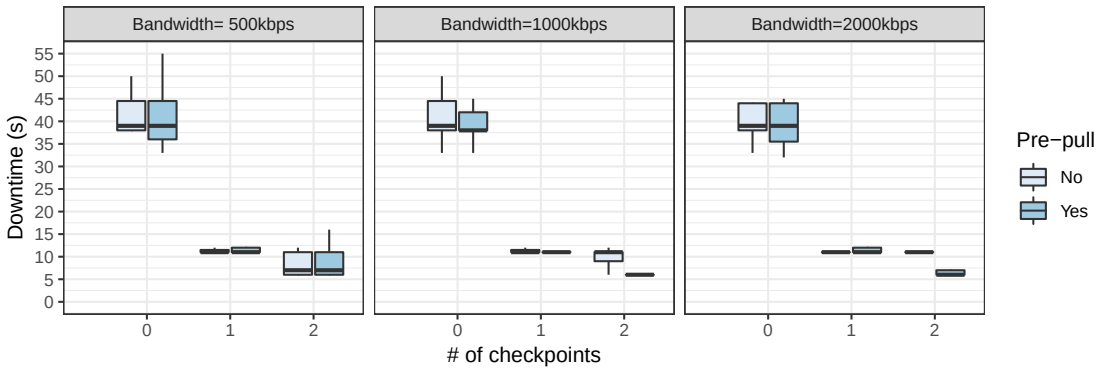

Fig. 7. Migration downtime.

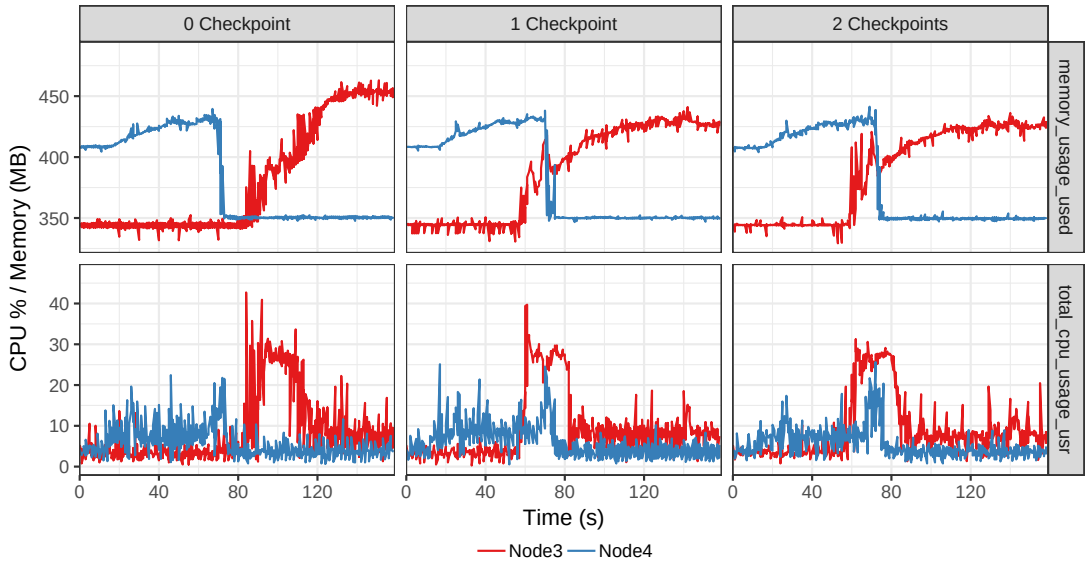

Fig. 8. CPU and memory usage during container migration.

This increased CPU usage is more important in the scenario with no checkpoint compared to one and two checkpoints, as more messages have been delayed due to a longer container downtime.

In all scenarios we notice that resource usage remains reasonable and not significantly different from that of a running container with no migration.

VI. Conclusion

Container migration is an essential functionality in large-scale geo-distributed platforms such as fog computing infrastructures. Contrary to migration within a single data center, long-distance migration requires that the container's disk state should be migrated together with the container itself. However, this state may be arbitrarily large, so its transfer may create long periods of unavailability for the container. We proposed to exploit the layered structure provided by the OverlayFS file system to transparently snapshot the volumes' contents and transfer them prior to the actual container migration. We implemented this mechanism within Kubernetes and demonstrated that it reduces the container's downtime during migration by a factor 4 compared to a baseline with no volume checkpoint.

References

[1] Rajesh K, "Importance of virtual machine migration in server virtualization," excITingIP.com, 2011, https://bit.ly/3ih5lQR.

[2] A. Fahs and G. Pierre, "Proximity-aware traffic routing in distributed fog computing platforms," in *Proc. ACM/IEEE CCGrid*, 2019.

[3] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *Proc. IEEE/IFIP NOMS*, 2018.

[4] M. Chima Ogbuachi, A. Reale, P. Suskovics, and B. Kovacs, "Context-aware Kubernetes scheduler for edge-native applications on 5G," *Journal of Communications Software and Systems*, vol. 16, no. 1, 2020.

[5] J. Santos, T. Wauters, B. Volckaert, and P. De Turck, "Towards network-aware resource provisioning in Kubernetes for fog computing applications," in *Proc. NetSoft*, 2019.

[6] W.-S. Zheng and L.-H. Yen, "Auto-scaling in Kubernetes-based fog computing platform," in *Proc. ICS*, 2018.

[7] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, 2019.

[8] B. Salmon, "Understanding cloud storage models," InfoWorld, 2015, https://bit.ly/2VCLt0V.

[9] S. J. Vaughan-Nichols, "What is Docker and why is it so darn popular?" ZDNet, 2018, https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, 2016.

[11] L. Abdollahi Vayghan, M. Aymen Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with Kubernetes: Experiments and lessons learned," in *Proc. IEEE CLOUD*, 2018.

[12] The Kubernetes authors, "Volumes," https://kubernetes.io/docs/concepts/storage/volumes/.

[13] W. Bao, D. Yuan, Z. Yang, S. Wang, W. Li, B. B. Zhou, and A. Y. Zomaya, "Follow Me Fog: Toward seamless handover timing schemes in a fog computing environment," *IEEE Communications Magazine*, vol. 55, no. 11, 2017.

[14] I. Farris, T. Taleb, A. Iera, and H. Flinck, "Lightweight service replication for ultra-short latency applications in mobile edge networks," in *Proc. IEEE ICC*, 2017.

[15] N. B. Truong, G. M. Lee, and Y. Ghamri-Doudane, "Software defined networking-based vehicular adhoc network with fog computing," in *Proc. IFIP/IEEE IM*, 2015.

[16] OpenVZ team, "CRIU," https://criu.org/.

[17] L. Deshpande and K. Liu, "Edge computing embedded platform with container migration," in *Proc. IEEE SmartWorld*, 2017.

[18] S. Oh and J. Kim, "Stateful container migration employing checkpoint-based restoration for orchestrated container clusters," in *Proc. ICTC*, 2018.

[19] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *Proc. IEEE ICDCS*, 2017.

[20] M. Sindi and J. R. Williams, "Using container migration for HPC workloads resilience," in *Proc. IEEE HPEC*, 2019.

[21] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory CRIU for Docker containers," in *Proc. MEMSYS*, 2019.

[22] A. Ahmed, A. Mohan, G. Cooperman, and G. Pierre, "Docker container deployment in distributed fog infrastructures with checkpoint/restart," in *Proc. IEEE Mobile Cloud*, 2020.

[23] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via Docker container migration," in *Proc. ACM/IEEE SEC*, 2017.

[24] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, J. Yang, M. Annavaram, and M. Hsieh, "Performance analysis of containerized applications on local and remote storage," in *Proc. MSST*, 2017.

[25] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on Docker containerization over RaspberryPi," in *Proc. ACM ICDCN*, 2017.

[26] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud*, Apr. 2017.

[27] H. Arkian, D. Giouroukis, P. Souza Junior, and G. Pierre, "Potable water management with integrated fog computing and LoRaWAN technologies," *IEEE IoT Newsletter*, 2020.

[28] Pivotal, "RabbitMQ – messaging that just works," https://www.rabbitmq.com/.

[29] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. R. Souza Jr, M. Ayalew Tamiru, and L. Wu, "Fog computing applications: Taxonomy and requirements," 2019, http://arxiv.org/abs/1907.11621.

[30] Pivotal, "RabbitMQ PerfTest," https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/.

[31] S. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, 2020.